# c++编码规范

#### **文件名**

文件名须全部小写，下划线（\_）和短横线（-）

源文件采用 UTF-8 编码。

#### **非空代码块：K&R 风格**

左花括号不另起新行

左花括号后紧跟换行

右花括号前需要换行

如果右花括号结束了语句，或者它是函数、类、类中的方法的结束括号，则其后面需要换行。如果后面紧跟的是 else，catch 或 while，或逗号，分号以及右括号，则不需要跟一个换行。

#### **代码块中的编进：+2 个空格**

新开代码块中代码需要加 2 个空格的缩进

##### **数组字面量：类代码块（block-like）**

|  |  |
| --- | --- |
| 1  2  3  4  5 | const c = [0, 1, 2];    someMethod(foo, [    0, 1, 2,  ], bar); |

##### **Switch 语句**

一如其他代码块一样，switch 语句中每个条件块也是增加 2 个空格的缩进。 每个 switch 标签后新起一行，加缩进，就像是创建了一个带花括号的代码块一样。每个标签开始时缩进又恢复，相当于只有 switch 标签中的内容是当作代码块来处理的。

break 关键字后面的空行是可选的。

switch (animal) {

case Animal.BANDERSNATCH:

handleBandersnatch();

break;

case Animal.JABBERWOCK:

handleJabberwock();

break;

default:

throw new Error('Unknown animal');

}

#### **声明语句**

##### **一个声明占一行**

每个声明语句后都跟换行。

##### **分号是必需的**

语句后需用分号结束。不能依赖于编辑器的自动分号插入功能。

#### **最大列宽：80**

##### **何处该换行**

换行的的准则是：尽量在优先级高的语法层面（higher syntactic level）进行

currentEstimate =

calc(currentEstimate + x \* currentEstimate) /

2.0f;

语法优先级从高到低依次为：赋值，除法，函数调用，参数，数字常量。

##### **换行后后续行至少有 4 个空格的缩进**

当发生换行时，第一行后面跟着的其他行至少缩进 4 个空格，除非满足代码块的缩进规则，另说。

换行后后续跟随多行时，缩进可适当大于 4 个空格。通常，语法中低优先级的后续行以 4 的倍数进行缩进，如果只有两行并且处于同一优先级，则保持一样的缩进即可。

#### **空格**

##### **垂直方向的空格**

以下场景需要有一个空行：

类或对象中的方法间 a. 例外的情形：对象中属性间的空行是可选的。如果有的话，一般是用来将属性进行分组。

方法体中，尽量少地使用空行来进行代码的分隔。函数体开始和结束都不要加空行。 3.类或对象中首个方法前及最后一个方法后的空行，既不提倡也不反对。

适用此风格中其他条目的规定（e.g.[3.4 goog.require statements](https://google.github.io/styleguide/jsguide.html" \l "file-goog-require)）。

连续多个空行不是必需的，但了不鼓励这么做。

##### **水平方向的空格**

水平方向的空格依位置为定，有三种大的分类：行首（一行的开始），行尾（一行的结束）以及行间（一行中除去行首及行尾的部分）。行首的空格（i.e. 缩进）无处不在。行尾的空格是禁止的。

除了 Javascript 本身及其他规则的要求，还有字面量，注释，JSDoc 等需要的空格外，单个的 ASCII 类型的空格在以下情形中也是需要的。

将关键字（譬如 if，for，catch）与括号（(）分隔。

将关键字（else，catch）与闭合括号（}） 分隔。

对于左花括号有两种例外： a. 作为函数首个参数的对象之前，数组中首个对象元素 （e.g. foo({a: [{c: d}]})） b. 在模板表达式中，因为模板语法的限制不能加空格（e.g. abc${1 + 2}def）。

二元，三元操作符的两边

逗号或分号后。但其前面是不允许有空格的。

对象字面量中冒号后面。

双斜线（//）两边。这里可以使用多个空格，但也不是必需的。

JSDoc 注释及其两边 e.g. 简写的类型声明 this.foo = /\*\* @type {number} \*/ (bar); 或类型转换（cast）function(/\*\* string \*/ foo) {。

#### **函数参数**

推荐将函数的所有参数放在与函数名同一行的位置。如果这样会导致列宽超限，那参数应该以一种易读的方式进行换行。为了节省空间，尽量超过宽度限制时才进行换行，换行后每个参数一行以提高可读性。换行后缩进为 4 个空格。与括号对齐是可以的，但不推荐。以下是觉的参数换行示例：

// Arguments start on a new line, indented four spaces. Preferred when the

// arguments don't fit on the same line with the function name (or the keyword

// "function") but fit entirely on the second line. Works with very long

// function names, survives renaming without reindenting, low on space.doSomething(

descriptiveArgumentOne, descriptiveArgumentTwo, descriptiveArgumentThree) {

// …}

// If the argument list is longer, wrap at 80. Uses less vertical space,

// but violates the rectangle rule and is thus not recommended.doSomething(veryDescriptiveArgumentNumberOne, veryDescriptiveArgumentTwo,

tableModelEventHandlerProxy, artichokeDescriptorAdapterIterator) {

// …}

// Four-space, one argument per line. Works with long function names,

// survives renaming, and emphasizes each argument.doSomething(

veryDescriptiveArgumentNumberOne,

veryDescriptiveArgumentTwo,

tableModelEventHandlerProxy,

artichokeDescriptorAdapterIterator) {

// …

}

#### **注释**

##### **块状注释**

块状注释与被注释代码保持相同缩进。/\* ... \*/ 和 // 都是。对于多行的 /\* ... \*/ 注释，后续注释行以 \* 开头且与上一行缩进保持一致。参数的注释紧随参数之后，用于在函数名或参数名无法完全表达其意思的情况。

注意区分 JSDoc（/\*\* ... \*/），上面不注释不能写成这样。

#### **本地变量的声明**

##### **使用 const 和 let**

所有本地变量使用 const 或 let 来声明。默认使用 const，除非该变量需要重新赋值。杜绝使用 var。

##### **一次声明一个变量**

一次只声明一个本地变量，let a = 1, b = 2; 这样的做法是禁止的。

##### **需要时才声明，声明后尽快初始化**

本地变量不要全部一次性声明在代码块的开头。声明在该变量每一次需要被使用的地方，以减少其影响范围。

##### **尽量标明类型**

JSDoc 类型注释可以添加在声明语句上面，或者内联到变量名前面。

const /\*\* !Array<number> \*/ data = [];

/\*\* @type {!Array<number>} \*/

const data = [];

#### **数组字面量**

##### **尾部加逗号**

始终在元素后面带上一个结束的逗号。

##### **属性引号保持一致(字典键的类型一致)**

对象属性分为不包含引号或 Symbols 的 structs 类型和包含引号或运算属性的 dicts 类型。两种类型不要混合使用，应该保持一致。

#### **标识符类型的命名规则**

##### **包名**

包名使用小写开头的驼峰命名 lowerCamelCase

##### **类名**

类，接口，record (这是什么？) 以及 typedef names (类型定义符) 使用大写开头的驼峰 UpperCamelCase。

未被导出的类只本地使用，并没有用 @private 标识，所以命名上不需要以下划线结尾。

类型名称通常为名词或名词短语。比如，Request，ImmutableList，或者 VisibilityMode。此外，接口名有时会是一个形容词或形容短语（比如 Readable）。

##### **方法名**

方法名使用小写开头的驼峰。私有方法需以下划线结尾。

方法名一般为动词或动词短语。比如 sendMessage 或者 stop\_。属性的 Getter 或 Setter 不是必需的，如果有的话，也是小写驼峰命名且需要类似这样 getFoo(对于布尔值使用 isFoo 或 hasFoo 形式)，setFoo(value)。

注：私有属性或方法以下划线开头才是主流吧

单元测试代码中的方法名会出现用下划线来分隔组件形式。一种典型的形式是这样的test<MethodUnderTest>\_<state>，例如 testPop\_emptyStack。对于这种测试代码中的方法，命名上没有统一的要求

##### **枚举**

枚举使用大写开头的驼峰，和类相似，一般一个单数形式的名词。枚举中的元素写成 CONSTANT\_CASE。

##### **常量**

常量写成 CONSTANT\_CASE：所有字母使用大写，以下划线分隔单词。私有静态属性可以用内部变量代替，所以不会有使用私有枚举的情况，也就无需将常量以下划线结尾来命名。

###### **常量的定义**

每个常量都是 @const 标识的静态属性或 模块内部通过 const 声明的变量，但并不是所有 @const 标识的静态属性或 const 声明的变量都是常理。需要常量时，先想清楚该对象是否真的不可变。例如，如果该对象中做生意状态可被改变，显然不适合作为常量。只是想着不去改变它的值是不够的，我们要求它需要从本质上来说应该一成不变。

##### **非常量字段**

非常量字段（静态或其他）使用小写开头的驼峰，如果是私有的私有的则以下划线结尾。

一般是名词或名词短语。例如 computedValues，index\_。

##### **参数**

参数使用小写开头的驼峰。即使参数需要一个构造器来初始化时，也是这一规则。

公有方法的参数名不能只使用一个字母。

**例外：**如果三方库需要，参数名可以用 $ 开头。此例外不适用于其他标识符（e.g. 本地变量或属性）。

##### **本地变量**

本地变量使用小写开头的驼峰，除非是上面介绍过的本地常量。函数中的常量命名仍然使用小写开头的驼峰lowerCamelCase。即使该变量指向的是构造器也使用 lowerCamelCase。

##### **模板参数/Template parameter names**

模板参数力求简洁，用一个单词，一个字母表示，全部使用大写，例如 TYPE，THIS。

#### **驼峰：定义**

有时将一个英文短语转成驼峰有很多形式，例如首字母进行缩略，IPv6 以及 iOS 这种都有出现。为保证代码可控，本规范规定出如下规则。

将短语移除撇号转成 ASCII 表示。例如 Müller's algorithm 表示成 Muellers algorithm。

将上述结果拆分成单词，以空格或其他不发音符号（中横线）进行分隔。 a. 推荐的做法：如果其中包含一个已经常用的驼峰翻译，直接提取出来（e.g. AdWords 会成为 ad words）。需要注意的是 iOS 本身并不是个驼峰形式，它不属性任何形式，所以它不适用本条规则。

将所有字母转成小写，然后将以下情况中的首字母大写： a. 每个单词的首字母，这样便得到了大写开头的驼峰 b. 除首个单词的其他所有单词的首字母，这样得到小写开头的驼峰

将上述结果合并。

过程中原来名称中的大小写均被忽略。

示例：

| **Prose form** | **Correct** | **Incorrect** |
| --- | --- | --- |
| XML HTTP request | XmlHttpRequest | XMLHTTPRequest |
| new customer ID | newCustomerId | newCustomerID |
| inner stopwatch | innerStopwatch | innerStopWatch |
| supports IPv6 on iOS? | supportsIpv6OnIos | supportsIPv6OnIOS |
| ouTube importer | YouTubeImporter | YoutubeImporter\* |

\*这种情况可接受，但不推荐。

**注释：一些英文词汇通过中横线连接的方式是有歧义的，比如 "nonempty" 和 "non-empty" 都是正确写法，所以方法名 checkNonempty checkNonEmpty 都算正确。**

### **JSDoc**

[JSDoc](https://developers.google.com/closure/compiler/docs/js-for-compiler) 使用在了所有的类，字段以及方法上。

#### **通用形式**

JSDoc 基本的形式如下：

/\*\*

\* Multiple lines of JSDoc text are written here,

\* wrapped normally.

\* @param {number} arg A number to do something to.

\*/function doSomething(arg) { … }

或者这种单行的形式：

/\*\* @const @private {!Foo} A short bit of JSDoc. \*/this.foo\_ = foo;

如果单行形式长到需要折行，则需要切换到多行模式而不是使用单行形式。

有许多工具会对 JSDoc 文档进行解析以提取出有效的信息对代码进行检查和优化。所以这些注释需要好好写。

#### **Markdown**

JSDoc 支持 Markdown，所以必要时可包含 HTML。

工具会自动提取 JSDoc 的内容，其中自己书写的格式会被忽略。比如如果你写成下面这个样子：

/\*\*

\* Computes weight based on three factors:

\* items sent

\* items received

\* last timestamp

\*/

最终提取出来是这样的：

Computes weight based on three factors: items sent items received last timestamp

取而代之，我们应该按 markdown 的语法来格式化：

#### **JSDoc tags**

本规则可使用 JSDoc tags 的一个子集。详细列表见 [9.1 JSDoc tag reference](https://google.github.io/styleguide/jsguide.html" \l "appendices-jsdoc-tag-reference)。大部分 tags 独占一行。

错误的示例：

/\*\*

\* The "param" tag must occupy its own line and may not be combined.

\* @param {number} left @param {number} right

\*/function add(left, right) { ... }

简单的 tag 无需额外数据（比如 @private，@const，@final，@export），可以合并到一行。

/\*\*

\* Place more complex annotations (like "implements" and "template")

\* on their own lines. Multiple simple tags (like "export" and "final")

\* may be combined in one line.

\* @export @final

\* @implements {Iterable<TYPE>}

\* @template TYPE

\*/

class MyClass {

/\*\*

\* @param {!ObjType} obj Some object.

\* @param {number=} num An optional number.

\*/

constructor(obj, num = 42) {

/\*\* @private @const {!Array<!ObjType|number>} \*/

this.data\_ = [obj, num];

}

}

关于合并及合并后的顺序没有明确的规范，代码中保持一致即可。

详细的类型信息可参见 [Annotating JavaScript for the Closure Compiler](https://github.com/google/closure-compiler/wiki/Annotating-JavaScript-for-the-Closure-Compiler) 和 [Types in the Closure Type System](https://github.com/google/closure-compiler/wiki/Types-in-the-Closure-Type-System)。

#### **换行**

换行之后的 tag 块使用四个空格进行缩进。

/\*\*

\* Illustrates line wrapping for long param/return descriptions.

\* @param {string} foo This is a param with a description too long to fit in

\* one line.

\* @return {number} This returns something that has a description too long to

\* fit in one line.

\*/

exports.method = function(foo) {

return 5;

};

@fileoverview 换行时不缩进。

#### **文件头部注释**

一个文件可以在头部有个总览。包括版权信息，作者以及默认可选的[可见信息/visibility level](https://google.github.io/styleguide/jsguide.html" \l "jsdoc-visibility-annotations)等。文件中包含多个类时，头部这个总览显得很有必要。它可以帮助别人快速了解该文件的内容。如果写了，则应该有一个描述字段简单介绍文件中的内容以及一些依赖，或者其他信息。换行后不缩进。

示例：

/\*\*

\* @fileoverview Description of file, its uses and information

\* about its dependencies.

\* @package

\*/

#### **类的注释**

类，接口以及 records 需要有描述，参数，实现的接口以及可见性或其他适当的 tags 注释。类的描述需要告诉读者类的作用及何时使用该类，以及其他一些可以帮助别人正确使用该类的有用信息。构造器上的文本描述可省略。@constructor 和 @extends 不与 class 一起使用，除非该类是用来声明接口 @interface 或者扩展一个泛型类。

/\*\*

\* A fancier event target that does cool things.

\* @implements {Iterable<string>}

\*/

class MyFancyTarget extends EventTarget {

/\*\*

\* @param {string} arg1 An argument that makes this more interesting.

\* @param {!Array<number>} arg2 List of numbers to be processed.

\*/

constructor(arg1, arg2) {

// ... }

};

/\*\*

\* Records are also helpful.

\* @extends {Iterator<TYPE>}

\* @record

\* @template TYPE

\*/

class Listable {

/\*\* @return {TYPE} The next item in line to be returned. \*/

next() {}

}

#### **枚举和 typedef 注释**

枚举和 typedef 需要写文档。仅有的枚举和 typedef 其文档的描述不能为空。枚举中单个元素的文档可直接写在元素的前面一行。

/\*\*

\* A useful type union, which is reused often.

\* @typedef {!Bandersnatch|!BandersnatchType}

\*/

let CoolUnionType;

/\*\*

\* Types of bandersnatches.

\* @enum {string}

\*/

const BandersnatchType = {

/\*\* This kind is really frumious. \*/

FRUMIOUS: 'frumious',

/\*\* The less-frumious kind. \*/

MANXOME: 'manxome',

};

Typedefs 可方便地用于定义 records 类型，或 unions 的别名，复杂函数，或者 泛型类型。Typedefs 不适合用来定义字段很多的 records，因为其不支持对每个字段进行文档书写，也不适合用于模板或递归引用中。对于大型 records 使用 @record。

#### **方法与函数**

参数和返回类型需要写文档。必要时 this 也需要在文档中说明。方法，参数及返回的描述在方法的其他 JSDoc 中或方法签名中有表述，那么也是可以省略的。方法的描述应使用第三人称。如果方法重载父类中的方法，需要使用 @override 标识。重载方法需要包含所有的参数 @param 以及 @return 如果类型有变的话，如果没变也可省略。

/\*\* This is a class. \*/

class SomeClass extends SomeBaseClass {

/\*\*

\* Operates on an instance of MyClass and returns something.

\* @param {!MyClass} obj An object that for some reason needs detailed

\* explanation that spans multiple lines.

\* @param {!OtherClass} obviousOtherClass

\* @return {boolean} Whether something occurred.

\*/

someMethod(obj, obviousOtherClass) { ... }

/\*\* @override \*/

overriddenMethod(param) { ... }

}

/\*\*

\* Demonstrates how top-level functions follow the same rules. This one

\* makes an array.

\* @param {TYPE} arg

\* @return {!Array<TYPE>}

\* @template TYPE

\*/function makeArray(arg) { ... }

匿名函数不需要写 JSDoc，但在自动推荐参数类型有困难时，也可手动指定一下类型。

promise.then(

(/\*\* !Array<number|string> \*/ items) => {

doSomethingWith(items);

return /\*\* @type {string} \*/ (items[0]);

});

#### **属性**

属性的类型需要加文档。对于私有属性如果其命名已经很好地提示了其类型，则描述可省略。

公有的常量与属性是一样的文档写法。当 @const 类型的属性从一个表达式初始化时，是能明显看出其类型的，此时其显式的类型指定可省略。

**小贴士：@const 属性的类型从构造器的参数进行初始化时，其类型是“显然”的，因为参数上面声明了类型，或者从一个函数调用进行初始化，因为函数返回值的类型也是声明了的。对于非常量或者它的值来源不是很明朗的情况下才需要指定其类型。**

/\*\* My class. \*/

class MyClass {

/\*\* @param {string=} someString \*/

constructor(someString = 'default string') {

/\*\* @private @const \*/

this.someString\_ = someString;

/\*\* @private @const {!OtherType} \*/

this.someOtherThing\_ = functionThatReturnsAThing();

/\*\*

\* Maximum number of things per pane.

\* @type {number}

\*/

this.someProperty = 4;

}

}

/\*\*

\* The number of times we'll try before giving up.

\* @const

\*/

MyClass.RETRY\_COUNT = 33;

#### **警告**

##### **如何处理警告**

解决之前，先明白警告的意思。如果不清楚，可以询问。

一旦了解之后，可进行如下操作：

**首先，修复或找替代方案。** 尝试针对该警告进行修复，或者换种方式实现相同的功能同时规避掉警告。

**或者，检查看是否是一个误报。** 如果确认代码没问题，警告是误报，写下相应注释并添加 @suppress标识。

**或者，留下 TODO 注释。** 这是最次的做法。这种做法相当于直接忽略警告，直到时机成熟再解决。

##### **将警告控制在最小范围**

将警告控制在最小合理的作用域范围，通常一个本地变量或很小的方法范围内。然后可以将这个变量或方法单独提取出来。

示例：

/\*\* @suppress {uselessCode} Unrecognized 'use asm' declaration \*/function fn() {

'use asm';

return 0;

}

一个类中大量的 @suppress 也好过编译时报大量的警告。

#### **已有代码的重新格式化**

更新已有代码时，遵循以下原则：

没必要更新所有老代码以满足本规范。需要在成本与代码一致性之间找个平衡点。规范不断在演变，花大成本更新老代码需要折衷。然而，如果老文件大部分都被修改了的话，那可以顺便将其全部改为符合现在的规范。

注意控制改动范围。如果你发现需要投入大量精力去更新代码而影响了当前需求的进展，考虑将这些老代码的更新另起一个分支。

##### **新增代码：遵循本规范**

全新创建的文件应该全部遵循本规范，某些包中其他类型文件有其他规范的另说。

向一个不是遵循本规范而写的文件添加新代码时，推荐先重新格式化当前文件

如果重新格式化完不成，那么新加的代码应该与老代码尽量保持一致，但不要滥用规范。